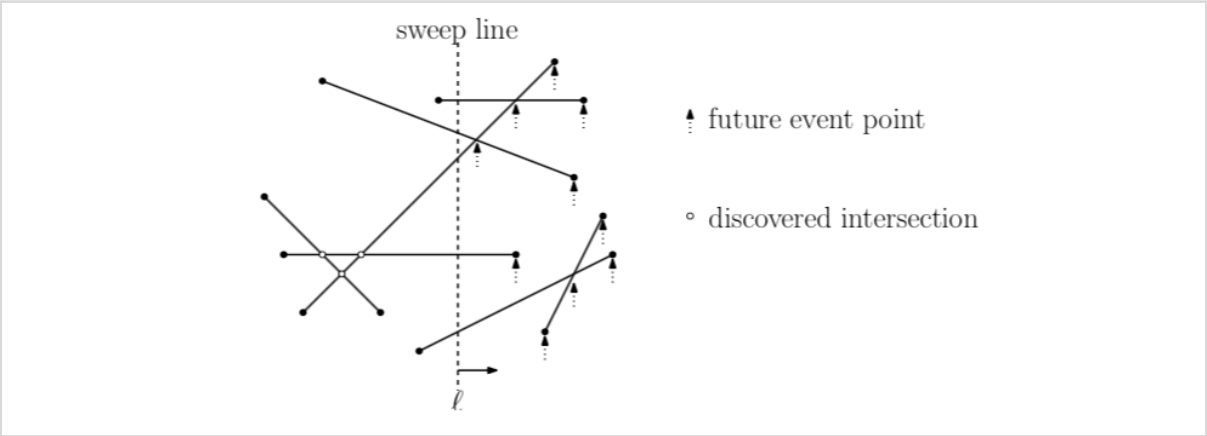
CS 6301.002.20S Lecture 4—January 23, 2020

Main topics are #planesweep_algorithms, #line_segment_intersection, and #doubly-connected_edge_lists.

Line Segment Intersection Continued

- Last Tuesday, we considered the following problem.
- We're given a set S of line segments in the plane. We want to report (output) every one of their intersections.
- Let n = |S| and I = the number of intersections.
- We'll discuss and algorithm of Bentley and Ottmann ['79] that runs in O((n + I) log n) time.
- The algorithm does a plane sweep. We sweep a vertical line across the plane from left to right. Certain important events occur when we sweep over the event points: the segment endpoints and intersections.



- The continuous sweeping is conceptual. The algorithm explicitly stores and updates just a few things:
  1. the partial solution of all intersection points to the *left* of the sweep line
  2. the current sweep line status: all line segments intersecting the sweep line with intersections ordered from top to bottom.
  3. a *subset* of future event points that need processed: all future segment endpoints and any intersections from adjacent segments on the sweep line

Data Structures

- For sweep line status, we need to know the line segments intersecting the sweep line in top to bottom order, and we need to do quick updates and useful queries
- For this reason, we'll store the status as an *ordered dictionary*.
- An ordered dictionary stores a bunch of ordered elements. There's lots of implementations including balanced binary search trees and skip lists.
- We need one that does the following operations:

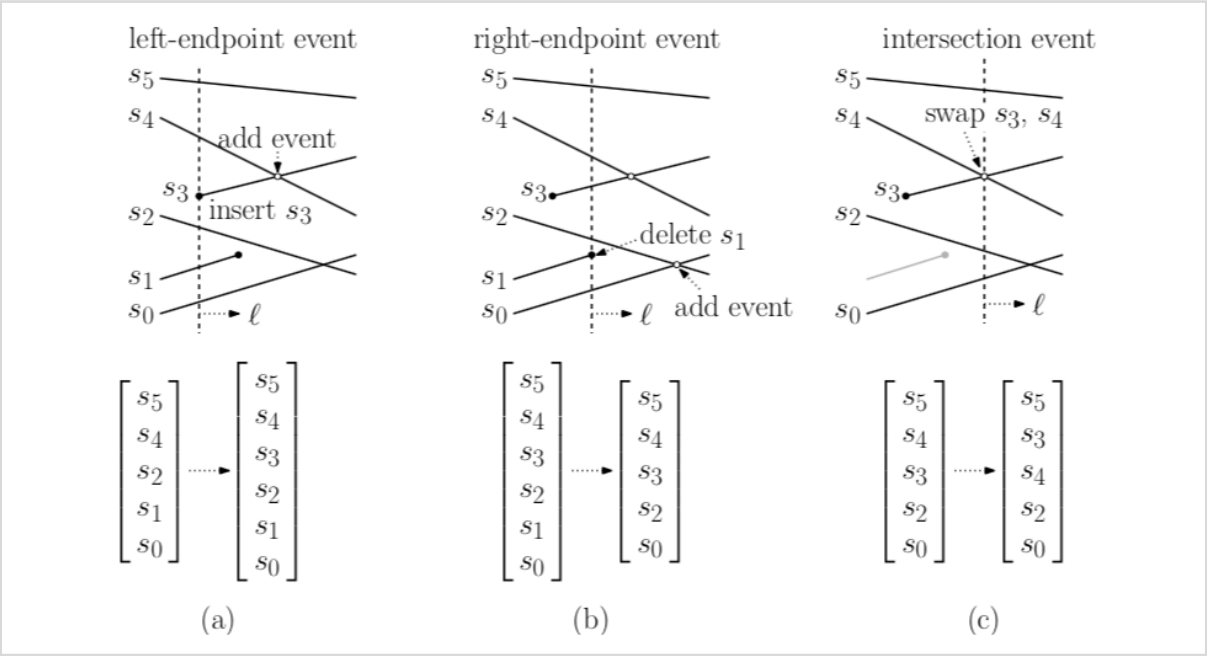
    - r ← insert(s): Insert object s and return a reference r to its location in the ordered dictionary.
    - delete(r): Delete entry r.
    - r' ← predecessor(r): Return a reference r' to object immediately before r (or null if r is the first object).
    - r' ← successor(r): Return a reference r' to object immediately after r (or null if r is last object).
    - r' ← swap(r): Swap r and its immediate successor returning a reference r' to its new location.
- With something like a balanced binary search tree, you can store m items in O(m) space and do these operations in O(log m) time each.
- But like when we were doing Jarvis's march, we do need to specify how to do comparisons before we can use an off-the-shelf ordered dictionary.
- Suppose we're sweeping the vertical line through (x_0, 0). To decide which of two line segments i and j is higher, we can express them with the normal line equations y= a_i x + b_i and y = a_j x + b_j. Then line i is higher than j iff a_i x_0 + b_i > a_j x_0 + b_j.

- We'll use a *priority queue* to store the event queue. We need one that supports the following operations:
    - r ← insert(e, x): Insert event e with priority x and return a reference r to its location in the priority queue.
    - delete(r): Delete the entry associated with reference r.
    - (e, x) ← extract-min(): Extract and return the event from the queue with smallest priority x.
- In our setting, the priority will just be the x-coordinate of the event's event point.

Processing Events

- Now we just need to process events. For this, it's probably just easiest for me to write out the algorithm and explain each step as I go.
- SegmentIntersections(S):
    - Insert segment endpoints into event queue.
    - While event queue is non-empty, extract the next event point p = (p_x, p_y).
        - If p is the left endpoint of segment s:
            - Insert s into sweep-line status for line at p_x.
            - Let s' and s'' be immediately above and below s.
            - If there is an event for intersection of s' and s'', remove it from event queue.
            - If s and s' intersect, add intersection to event queue.
            - If s and s'' intersect, add intersection to event queue.

- If p is the right endpoint of segment s.
  - Let s′ and s″ be immediately above and below s.
  - Delete s from sweep-line status.
  - If s′ and s″ intersect, add intersection to event queue.
- If p is an intersection between s′ and s″:
  - Report intersection of s′ and s″.
  - Swap s′ and s″ in sweep-line status.
  - Remove old events involving s′ and s″ from event queue.
  - Add new intersection events between s′ and s″ and their new respective neighbors.

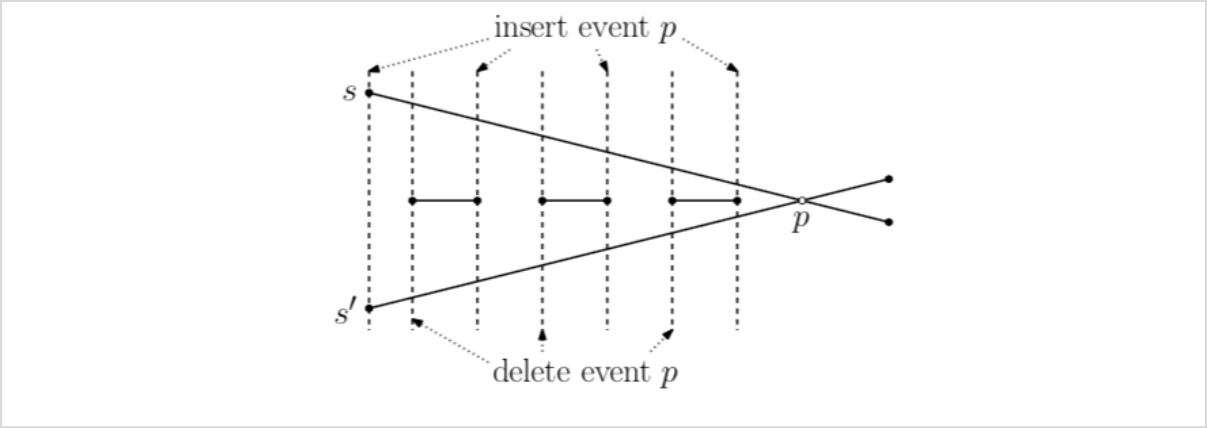


Analysis

- The sweep-line status has ≤ n segments at all times. The event queue has O(n) events at any time.
- Therefore, events take O(log n) time to process.
- I was the number of intersections. There are 2n + I events processed, so the total running time is O((n + I) log n).
- There is also an O(n log n + I) time algorithm I won't be giving in this class, as well as an Omega(n log n + I) lower bound.

Event Deletion

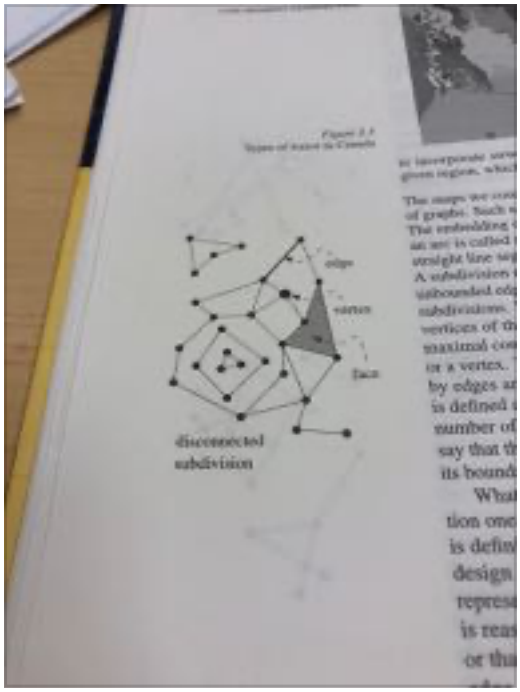
- One final note:
- Why did I find it necessary to delete events when line segments were no longer adjacent? Because you may end up adding the same event Omega(n) times otherwise!



- You could also handle this issue by trying to detect if your event queue already has an event like the book does, but I think that's messier.

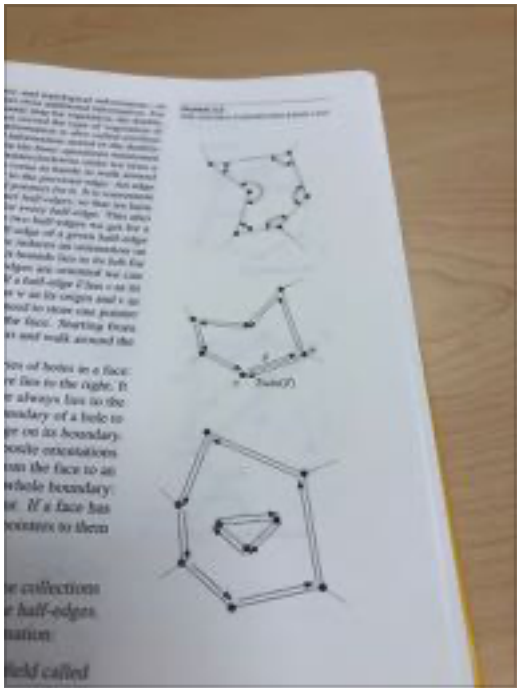
Planar Subdivisions and Doubly-connected Edge Lists

- For the rest of today, I want to introduce a data structure for *planar subdivisions* that we'll use throughout the semester.
- In particular, this data structure is used in an application of line segment intersection detailed in your textbook. I'll briefly touch upon the application, but won't focus on it. The data structure will also be useful for what we focus on next week.
- A *planar graph* is one where we can map vertices to points in the plane and edges to internally disjoint line segments between their vertex endpoints. The embedding breaks the plane into a number of maximally connected regions called *faces*. The planar subdivision is this combination of vertices, edges, and faces in the plane. The graph need not be connected, and the faces may have holes.

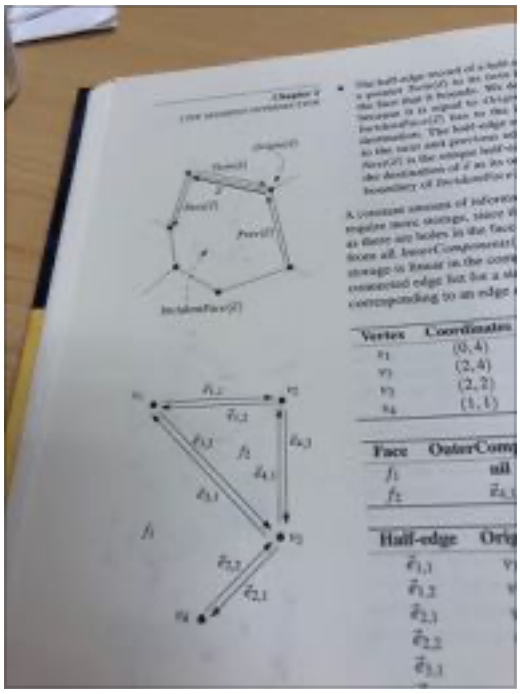


- We can represent planar subdivisions using something called a doubly-connected edge list or DCEL for short. This is a natural generalization of adjacency lists for general graphs.

- Each edge is incident to two faces (or maybe the same face twice), so we treat each edge as a pair of *twin* directed *half-edges*, each with an origin or tail and a destination or head. The origin of one half-edge is the destination of the other and vice versa.
- You can imagine half-edges going around faces counterclockwise, so we also store for each half-edge the next and previous half-edges along each face.
- There's also some info for getting between vertices and faces and their incident edges.



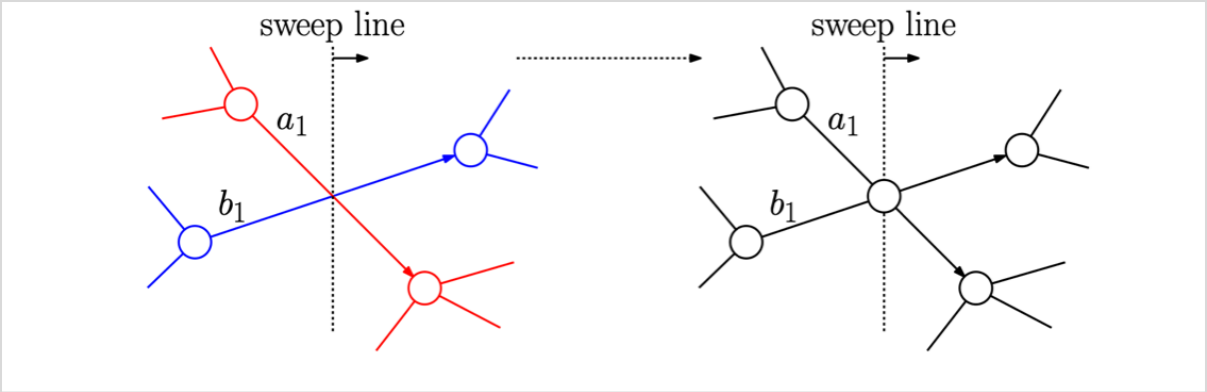
- Here's what we store for each vertex, edge, and face.
- For each vertex v:
  - Coordinates(v)
  - IncidentEdge(v): an arbitrary half-edge with v as the origin.
- For each face f:
  - OuterComponent(f): an arbitrary half-edge on the outer boundary of f with f on its left (null if f is the outer face).
  - InnerComponents(f): a *list* of half-edges with f on their left, one per hole in f.
- For each half-edge e->:
  - Origin(e->)
  - Twin(e->)
  - IncidentFace(e->): face to left of e->
  - Next(e->): next half-edge IncidentFace(e->)
  - Prev(e->)



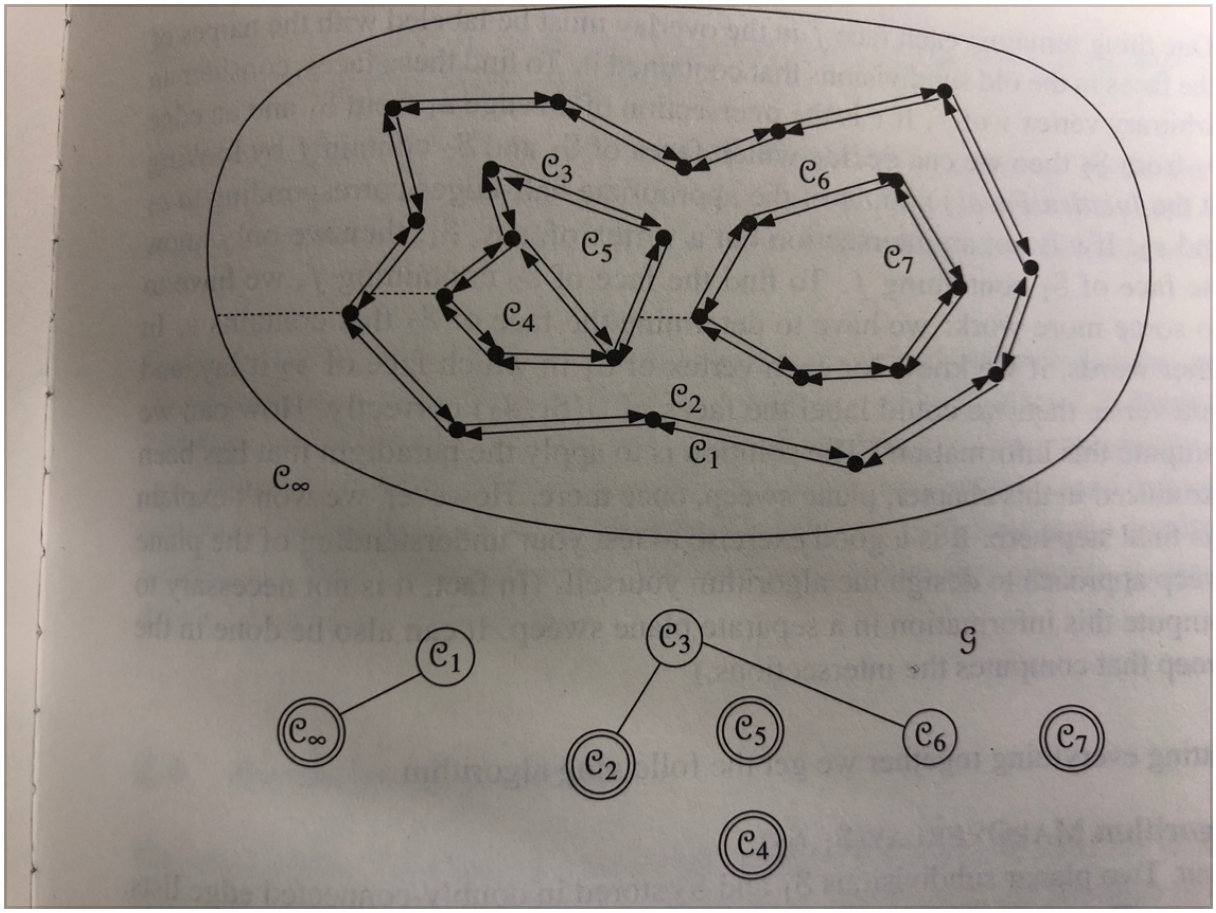
- With that data structure you can do pretty much any “local” thing in constant time like finding an incident face to an edge or vertex, traversing one edge along a face, traversal one edge around a vertex, or as we’ll see, even subdividing an edge with a new vertex or subdividing a face by adding a new edge.
- Usually, I won’t go into detail on how to use these individual values or refer to them by name. Just trust that most “reasonable" operations on a DCEL can be done in constant time.

Computing Overlays

- Here’s an application of both line segment intersection and DCELs.
- We’re given two planar subdivisions G_1 and G_2. We want to compute their *overlay*, the planar subdivision you get when you overlap all edges of both G_1 and G_2.
- Note I’m going to be brief here. I just want to give you a taste of what we accomplished with line segment intersections and what we can do with DCELs. We’ll go into other examples in much more detail over the coming weeks. The book has the full details.
- We compute the overlay in two phases, both aided by the line segment intersection plane sweep.
- First, we compute everything stored for the vertices and edges except the IncidentFace pointers. This includes the Next and Prev points. To do so, we put all of this information already known for G_1 and G_2 into one big DCEL G, ignoring that some edges overlap.
- Now we run our line segment intersection algorithm. At each intersection we encounter, we subdivide the intersecting edges by adding a new vertex.



- We're just deleting and inserting things into some lists, so this process takes constant time per intersection.
- After computing all the vertices and edges for the overlay, we still need to figure out the faces and link them to edges and vice versa.
- We can compute each facial cycle by following Next pointers. From there, we can easily compute the IncidentFace pointers.
- Computing the inner and outer components for each face is more subtle.
- The outer faces are those where there is an angle of less than 180° when traversing the highest vertex along the facial cycle where the face lies to the left of the cycle.
- To figure out what other cycles belong to each face, imagine a graph consisting of one node per facial cycle. Two nodes are adjacent if you can take a line segment directly from the highest vertex of a hole node to some edge on the other node without leaving the face. (this figure from the book uses line segments going left from leftmost vertices)



- Each connected component of the graph contains precisely the facial cycles/components for a single face.
- You'll know if an edge exists in the graph, because we already figured out which line segments lie directly above vertices while performing the line segment intersection sweep!